To appear in: *Journal of Functional Programming.*

On Cubism

BART JACOBS
CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

Abstract

A number of difficulties in the formalism of *Pure Type Systems* (PTS's) is discussed and an alternative classification system for typed calculi is proposed. In the new approach the main novelty is that one first explicitly specifies the dependencies that may occur. This is especially useful in order to describe constants. But it also facilitates the description of other type theoretic features, like dependent sums.

Capsule review

The paper reexamines the classification of typed lambda calculi as pure type systems, put forward by Barendregt. A problematic aspect of the PTS-framework is the incorporation of constants: this can be done, but at the cost of ad hoc extensions with products. The paper proposes an alternative description of type systems, essentially by enriching PTS's, with settings plus features, obtaining a neat mechanism for handling constants.

1 Introduction

The phrase *cubism* will be used for the school that advocates classification of typed λ-calculi in terms of *Pure Type Sytems* (PTS's). Some of these can be arranged nicely in what has become known as *Barendregt's cube*. See [1] for an overview. This classification has both technical and conceptual defects, as I will argue below.

In my thesis [5] I studied categorical semantics of various typed calculi. It turned out that the formalism of PTS's wasn't really helpful in understanding the semantics of these systems. Instead, the notion of *dependence* (to be explained below) proved to be more fundamental and useful. It gives the possibility to "read off" the corresponding semantical structure almost directly, since this relation of dependence corresponds to the categorical notion of *fibred over*. Here I will not go into these semantical matters but will argue that there are good reasons within syntax itself for taking this notion of dependence as the starting point in the classification of typed calculi. The essentials of the alternative classification of typed calculi (as proposed in [5]) will be described here. In a nutshell, a new level is introduced which comes before what is specified in PTS's. In this new level one lays down which dependencies are allowed (in the system that one wishes to describe).

Typed calculi nowadays are not as simple any more as in the early days of Church [2]. They often involve different syntactic categories[1] or universes (called sorts in PTS's) like Type, Prop, Kind or Set. Thus it has become an important issue to classify such calculi and describe them in a uniform and systematic way. This is achieved in PTS's, which will be sketched first. Subsequently the alternative classification in terms of settings and features will be described. These two aproaches are then compared in a discussion.

2 Pure type systems

A pure type system (PTS) is given by three sets of *sorts*, *axioms* and *rules*. The axioms are of the form $\vdash c\colon s$ where s is a sort and c is a constant or a sort. A rule is a triple (s_1, s_2, s_3) of sorts which allows the following product formation rule.

$$\frac{\Gamma \vdash A\colon s_1 \qquad \Gamma, x\colon A \vdash B\colon s_2}{\Gamma \vdash \Pi x\colon A.\, B\colon s_3}\,(s_1, s_2, s_3)$$

(Mostly one has $s_2 = s_3$; in that case one simply writes (s_1, s_2) for this rule.) Such a PTS generates a calculus of expressions $\Gamma \vdash A\colon s$ (for s a sort) and $\Gamma \vdash M\colon A$, using the start rule

$$\frac{\Gamma \vdash A\colon s}{\Gamma, x\colon A \vdash x\colon A}$$

together with the axioms, and application and abstraction rules (plus a weakening and a conversion rule, see [1] for details). As usual one writes $A \to B = \Pi x\colon A.\, B$ if x is a variable which does not occur in B.

Particular examples of PTS's are given with two sorts, which we write as Type and Kind here ($*$ and $\Box$ in the PTS-world) with one axiom Type: Kind and the following rules.

System	**Rules**			
$\lambda\!\to$	(Type, Type)			
$\lambda 2$	(Type, Type)	(Kind, Type)		
$\lambda\underline{\omega}$	(Type, Type)		(Kind, Kind)	
$\lambda\omega$	(Type, Type)	(Kind, Type)	(Kind, Kind)	
λP	(Type, Type)			(Type, Kind)
$\lambda P2$	(Type, Type)	(Kind, Type)		(Type, Kind)
$\lambda P\underline{\omega}$	(Type, Type)		(Kind, Kind)	(Type, Kind)
λC	(Type, Type)	(Kind, Type)	(Kind, Kind)	(Type, Kind)

[1] Categories in the sense of philosophical logic, not as in category theory.

These systems can be organised elegantly in the form of a cube,

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\rightarrow} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

The arrows should be read as inclusions starting from the simplest calculus $\lambda{\rightarrow}$ in the lower left corner. This cube gives a possible fine-structure of the calculus of constructions λC (from [3]). But there are other ways to decompose λC. This will be described next.

3 Type systems as settings plus features

The following serves as a motivation. Consider your favourite type system, which I assume to have a number of sorts (or syntactic categories) and some mechanisms for building types and terms. Abstract away from all particular details. What remains are sequents of the form

$$x_1\colon A_1, \ldots, x_n\colon A_n \vdash A\colon s \qquad \text{and} \qquad x_1\colon A_1, \ldots, x_n\colon A_n \vdash M\colon A$$

expressing that A is of sort s in context $x_1\colon A_1, \ldots, x_n\colon A_n$, and that M is a term which inhabits A, again in context $x_1\colon A_1, \ldots, x_n\colon A_n$. The point of view put forward here is that such sequents contain the first piece of information in the classification of type systems, namely the dependencies that may occur (in the specific type system that we are considering). Let's formulate what these dependencies are.

Assume two sorts s_1, s_2 in a typed calculus. We say that s_2 *depends on* s_1 (in this calculus) in case there are expressions

$$\Gamma \vdash A\colon s_1 \qquad \text{and} \qquad \Gamma, x\colon A \vdash B\colon s_2$$

where the variable x occurs free in B. In this case we write $s_2 \succ s_1$. The idea is that children of s_2 (namely $B\colon s_2$) may contain grandchildren of s_1 (in this case $x\colon A\colon s_1$). This notion of dependency is related to indexing[2]: one can think of B as a family $\{B(a)\colon s_2\}_{a:A}$ indexed by $A\colon s_1$.

Consider for example an index set I together with a collection of sets $\{X_i\}_{i\in I}$ indexed by I. Formally we can write this as

$$\vdash I\colon \mathsf{Set} \qquad \text{and} \qquad i\colon I \vdash X_i\colon \mathsf{Set}$$

[2]And hence to indexed and fibred categories.

This shows that sets indexed by sets requires a dependency $\mathsf{Set} \succ \mathsf{Set}$, expressing that sets may depend on sets.

What is analysed here as a fundamental property is now formulated in abstract form.

3.1. Definition. A *setting* consists of a set S of sorts together with a transitive relation $\succ$ of dependency on S.

A type system is said to have this setting $(S, \succ)$—or to be built upon this setting—if all occurring dependencies are allowed by the setting. This means that for each pair of (well-formed, derivable) sequents

$$\Gamma \vdash A\colon s_1 \qquad \text{and} \qquad \Gamma, x\colon A \vdash B(x)\colon s_2$$

with the variable $x\colon A$ occurring free in $B(x)$, we have $s_2 \succ s_1$ in the setting.

Now we can turn things around: instead of deriving a setting from a type system (by inspection of which dependencies actually occur), we start from a setting, and look at the type systems which have this setting. This is conceptually an important step in our approach.

Again, at this stage, we only look at dependencies and at nothing else. For example not at how these dependencies may arise.

3.2. Examples. (i) Assume we have a setting with one sort Type and the dependency $\mathsf{Type} \succ \mathsf{Type}$. In a calculus with this setting one may have a type $B(x)\colon \mathsf{Type}$ containing a term variable $x\colon A$, where A is itself a type (i.e. $A\colon \mathsf{Type}$). But this captures *type dependency* as in Martin-Löf's type theory [7]. A typical example is

$$\vdash \mathbb{N}\colon \mathsf{Type} \qquad \text{and} \qquad n\colon \mathbb{N} \vdash \mathsf{natlist}(n)\colon \mathsf{Type}$$

where the latter is the type of lists of natural numbers of length n. This describes type-indexed types like set-indexed sets above.

(ii) If we have a setting with two sorts $\mathsf{Type}, \mathsf{Kind}$ with the dependency $\mathsf{Type} \succ \mathsf{Kind}$, then one can have types $\sigma(\alpha)\colon \mathsf{Type}$ containing a variable $\alpha\colon A$ for a kind $A\colon \mathsf{Kind}$. This situation occurs in polymorphic calculi. A special case is $A = \mathsf{Type}$ with $\mathsf{Type}\colon \mathsf{Kind}$ an axiom.

One sees how these settings capture certain characteristics of type systems. And this constitutes the first step in the classification, as proposed here.

With settings one can also do some combinatorics. The next diagram gives the settings with one and two sorts. The arrows are inclusions. The encircled numbers have no meaning but are there for future reference. The setting of λC now occurs at the bottom (as ⑦). This gives an alternative fine-structure.

$$\begin{tikzcd}[column sep=tiny]
& (\mathsf{Type}, \succ = \emptyset)^{\textcircled{1}} \arrow[dl] \arrow[dr, shift left] \arrow[dr, shift right] & \\
(\mathsf{Type}, \mathsf{Type} \succ \mathsf{Type})^{\textcircled{2}} \arrow[d] \arrow[drr] & & (\mathsf{Kind}, \mathsf{Type}, \mathsf{Type} \succ \mathsf{Kind})^{\textcircled{3}} \arrow[d] \arrow[dll] \\
(\mathsf{Kind}, \mathsf{Type}, \begin{pmatrix} \mathsf{Type} \succ \mathsf{Kind} \\ \mathsf{Kind} \succ \mathsf{Kind} \end{pmatrix})^{\textcircled{4}} \arrow[dr] & & (\mathsf{Kind}, \mathsf{Type}, \begin{pmatrix} \mathsf{Type} \succ \mathsf{Kind} \\ \mathsf{Type} \succ \mathsf{Type} \end{pmatrix})^{\textcircled{5}} \arrow[dl] \\
& (\mathsf{Kind}, \mathsf{Type}, \begin{pmatrix} \mathsf{Type} \succ \mathsf{Kind} \\ \mathsf{Kind} \succ \mathsf{Kind} \\ \mathsf{Type} \succ \mathsf{Type} \end{pmatrix})^{\textcircled{6}} \arrow[d] & \\
& (\mathsf{Kind}, \mathsf{Type}, \begin{pmatrix} \mathsf{Type} \succ \mathsf{Kind} \\ \mathsf{Kind} \succ \mathsf{Kind} \\ \mathsf{Type} \succ \mathsf{Type} \\ \mathsf{Kind} \succ \mathsf{Type} \end{pmatrix})^{\textcircled{7}} &
\end{tikzcd}$$

The double arrow ① $\rightrightarrows$ ③ indicates that there are two inclusions, namely $\mathsf{Type} \mapsto \mathsf{Type}$ and $\mathsf{Type} \mapsto \mathsf{Kind}$.

Most of these settings underly well-known systems.

Number	**Setting of**
①	simply typed calculi (understood here without type variables)
②	dependently typed calculi (like Martin-Löf's [7], without universes)
③	polymorphic calculi (the left plane $\lambda\!\rightarrow, \lambda 2, \lambda\underline{\omega}, \lambda\omega$ of the cube)
④, ⑤	have not been studied separately
⑥	HML [8] and the theory of predicates [9, 10]
⑦	the right plane $\lambda P, \lambda P2, \lambda P\underline{\omega}, \lambda C$ of the cube

What remains of the cube is a single edge ③ $\rightarrow$ ⑦. Hence the cube becomes one-dimensional.

The above systems HML of Moggi and the theory of predicates of Pavlović with underlying setting ⑥ were conceived as the calculus of constructions λC with setting ⑦, but without the dependency $\mathsf{Kind} \succ \mathsf{Type}$. Moggi precludes this dependency in HML, because he wishes to have a compile-time part of kinds which is independent of a run-time part of types. The motivation of Pavlović comes from logic: he thinks of types as propositions and of kinds as sets and doesn't want sets to depend on proofs (i.e. on inhabitants of propositions). Both arguments make good sense and form the basis for sensible type theories. Thus the only way to understand these subsystems of λC is to look at the underlying settings. This cannot be done (directly) in a PTS-framework. In fact it is not clear at all how to describe these systems in a PTS-format. The same holds for Martin-Löf's type theory.

A setting as understood here comes equipped with some basic rules for manipulating contexts, like projection, weakening and substitution. A specific calculus is then layed down by specifying some additional 'features' that fit to the setting. These features are mechanisms for building new types and terms. The collection of features is open-ended: it includes axioms, products Π, sums Σ, exponents $\rightarrow$, constants, equality, quotients, subtypes, inductive types etcetera. What we mean by mechanism, is not only the formation rule, but also the associated introduction-, elimination-, conversion-, equality- and substitution-rules.

3.3. Definition. A *feature* is a mechanism for building new types and terms. It comes together with a set of required dependencies. This set may be empty.

A feature may be added to a certain setting if all its sorts and all its required dependencies exist in the setting. We then say that the feature fits to (or, is appropriate in, or is allowed by) the setting.

A *type system* is a setting together with a number of features fitting to the setting.

The definition is somewhat vague, but this cannot be avoided due to the open-endedness of the collection of features. The following table contains a number of well-known features together with their required dependencies.

Feature	**Required dependencies**
axiom $\vdash s_1 : s_2$	$s_1 \succ s_2$
(s_1, s_2)-product Π	$s_2 \succ s_1$
weak (s_1, s_2)-sum Σ	$s_2 \succ s_1$
strong (s_1, s_2)-sum Σ	$s_2 \succ s_1$, $s_2 \succ s_2$
very strong (s_1, s_2)-sum Σ	$s_2 \succ s_1$, $s_2 \succ s_2$, $s_1 \succ s_2$
s-exponent $\rightarrow$	–
s-cartesian product $\times$	–

Here we shall briefly discuss axioms, products Π, exponents $\rightarrow$ and cartesian products $\times$. The various sums occur in the next section. Also, constants may be found there.

(i) If we have an axiom $\vdash s_1 : s_2$, then we can use the start rule

$$\frac{\vdash s_1 : s_2}{\alpha : s_1 \vdash \alpha : s_1}$$

This yields a grandchild α of s_2 occurring in a child $\alpha : s_1$ of s_1. Hence we have a dependency $s_1 \succ s_2$. Thus the axiom $\vdash s_1 : s_2$ may only be used in a setting with this dependency $s_1 \succ s_2$.

(ii) The formation of dependent (s_1, s_2)-products

$$\frac{\Gamma \vdash A : s_1 \qquad\qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A .\, B : s_2}\,(s_1, s_2)$$

only really makes sense if x may occur free in B. That is, if the dependency $s_2 \succ s_1$ actually occurs. Hence we take this as a required dependency for these products. Adding such a product feature to a type system means besides the above formation rule, also adding the associated rules like abstraction (introduction), application (elimination), conversions, equations and behaviour under substitution.

(iii) For s-exponents one has a formation rule,

$$\frac{\Gamma \vdash A\colon s \qquad\qquad \Gamma \vdash B\colon s}{\Gamma \vdash A \to B\colon s}$$

One sees that this has no influence on the dependencies which occur. Hence s-exponents may be added to any type system (in which the sort s occurs).

These s-exponents involve $A \to B$ for A, B of the *same* sort s. Semantically one thinks of the arrow type $A \to B$ as the exponent of objects A, B in a category. This should not be confused with the exponent notation $C \to D \equiv \Pi x\colon C.\, D$ if x does not occur in D. This involves quite a different feature, namely (s_1, s_2)-products Π for some pair s_1, s_2 of possibly distinct sorts. The (categorical) interpretation of (s_1, s_2)-products is different from exponents. Only in case $s_1 = s_2$ the arrow $C \to D \equiv \Pi x\colon C.\, D$ (with x not in D) can be understood as an s_1-exponent object.

The situation for s-cartesian products,

$$\frac{\Gamma \vdash A\colon s \qquad\qquad \Gamma \vdash B\colon s}{\Gamma \vdash A \times B\colon s}$$

is exactly the same: there are no required dependencies.

Notice that with the required dependencies $s_2 \succ s_1$ for (s_1, s_2)-products and $s_1 \succ s_2$ for an axiom $\vdash s_1\colon s_2$, it becomes clear how to turn a specific PTS into a type system as understood here. Just take all the sorts, together with the required dependencies for the products and axioms occurring in the PTS. (One may have to take the transitive closure.) This yields the underlying setting. The features of the type system are then the products and axioms of the PTS. They are allowed by construction.

There is in general no way to turn a type system consisting of setting plus features into a PTS. The notion of type system is much richer. It includes for example Martin-Löf's type theory, which cannot be described as a PTS.

We conclude this section with an example how to set up a specific type system. Some further details of settings and features will be discussed in the next section.

3.4. Example. One starts a type system by making explicit how many sorts one wishes and how they should depend on each other. Suppose we want a type theoretic version of "logic over dependent type theory", i.e. a predicate logic over dependent types (or sets). In logical terms this means that we want set-indexed sets as in

$$\vdash I\colon \mathsf{Set} \qquad \text{and} \qquad i\colon I \vdash X_i\colon \mathsf{Set}$$

and predicates over such sets, i.e.

$$\vdash I\colon \mathsf{Set} \qquad \text{and} \qquad i\colon I \vdash R(i)\colon \mathsf{Prop}$$

The first requires a dependency $\mathsf{Set} \succ \mathsf{Set}$ and the second $\mathsf{Prop} \succ \mathsf{Set}$. Such a logic was described in [6] (and called dependent logic there), and also in [11] as an expressive version of the internal language of a topos.

Under a propositions-as-types (and sets-as-kinds) reading we have $\mathsf{Prop} = \mathsf{Type}$ and $\mathsf{Set} = \mathsf{Kind}$. This shows we are in setting ④.

Once the setting has been fixed, we can proceed to specify the features that we wish to use in our calculus. For example, we may want

$(\mathsf{Set}, \mathsf{Set})$-products/sums	to form dependent products $\Pi i{:}\, I.\, X_i$ and sums $\Sigma i{:}\, I.\, X_i$.
$(\mathsf{Set}, \mathsf{Prop})$-products/sums	for quantification $\forall i{:}\, I.\, \varphi$ and $\exists i{:}\, I.\, \varphi$.
Prop-exponents	for implication $\varphi \to \psi$
$\vdash \mathsf{Prop}{:}\, \mathsf{Set}$	for higher order quantification $\forall \alpha{:}\, \mathsf{Prop}.\, (\alpha \to \alpha)$

And some constants, like

$$\vdash \mathbb{N}{:}\, \mathsf{Set}, \qquad n{:}\, \mathbb{N} \vdash \mathsf{natlist}(n){:}\, \mathsf{Set}, \qquad n{:}\, \mathbb{N} \vdash \mathsf{Even}(n){:}\, \mathsf{Prop}$$

These constants may be put together in what is commonly called a signature. It may be clear that all of these features fit to the setting $\mathsf{Set} \succ \mathsf{Set}$, $\mathsf{Prop} \succ \mathsf{Set}$.

Semantically, the dependency $\mathsf{Prop} \succ \mathsf{Set}$ tells that in a model, propositions must be indexed by sets. This gives some basic skeleton. The features are then captured by some additional structures which can be put on top of this skeleton.

4 Discussion

The main claim put forward here is that settings are fundamental in the classification of typed calculi. Firstly, they give a certain order between various systems. Secondly, they form the basis for the description of features on top of the setting.

This claim will be further illustrated in discussing three specific aspects: constants, sums and logic. The setting-plus-features-approach will be contrasted with the PTS-approach.

Constants. In the previous section constants were already mentioned in passing. It turns out that the underlying setting of a type system immediately tells which constants are allowed, and which are not. This will be considered in some detail.

In mathematics one often finds phrases like: let $\mathsf{Mat}(n, m)$ be the set of $n \times m$ matrices (over some fixed field). This can be seen as a declaration

$$n{:}\, \mathbb{N}, m{:}\, \mathbb{N} \vdash \mathsf{Mat}(n, m){:}\, \mathsf{Set}$$

of a "constant in context", or a "parametrized constant". Since $\mathbb{N}{:}\, \mathsf{Set}$ the above constant $\mathsf{Mat}(n, m)$ can be used in a setting with $\mathsf{Set} \succ \mathsf{Set}$; it requires this dependency, just like an axiom feature $\vdash s_1{:}\, s_2$ requires a dependency $s_1 \succ s_2$. Thus, if one is defining a type system with this dependency $\mathsf{Set} \succ \mathsf{Set}$, then one knows that a constant like $\mathsf{Mat}(n, m)$ can be used. So it becomes clear that the first important step in describing a specific type system is indeed saying what the setting is. Then one can describe the constants in a signature, and say what the other features are.

Similar examples are found in predicate logic. Here one needs a dependency $\mathsf{Prop} \succ \mathsf{Set}$. Indeed, one describes a specific predicate logic by specifying a signature

$$\langle F_1, \ldots, F_n, R_1, \ldots, R_m \rangle$$

consisting of function symbols F_i and predicate symbols R_j. The latter may involve free variables $\vec{x}$, say with $x_i\colon \sigma_i$ where $\sigma_i\colon \mathsf{Set}$. Formally one can describe such an R as a constant in context,

$$x_1\colon \sigma_1, \ldots, x_k\colon \sigma_k \vdash R(x_1, \ldots, x_k)\colon \mathsf{Prop}$$

And such constants are allowed with the dependency $\mathsf{Prop} \succ \mathsf{Set}$ of predicate logic (and of polymorphic calculi). One sees how easily such parametrized constants are described. And the setting immediately tells us which ones are allowed and which ones are not. Such constants are crucial in the practical use of type theory. And in some systems—like in Martin Löf's—constants are needed to get off the ground.

The general formulation is as follows. A constant

$$x_1\colon A_1, \ldots, x_n\colon A_n \vdash c(x_1, \ldots, x_n)\colon s \qquad \text{where} \qquad A_i\colon s_i$$

may be added to a type system if the underlying setting has all the dependencies $s \succ s_i$.

The smoothness with which constants can be described is a great advantage of working with these settings.

These constants in context are very natural, but they cannot be described within the PTS-formalism, despite a number of attempts to fit them in. There is however an indirect way: one could add extra means of quantification and describe the above constant R as

$$\vdash R\colon \sigma_1 \to \cdots \to \sigma_k \to \mathsf{Prop}$$

or, in case one has that the sets σ_i may depend on each other, as

$$\vdash R\colon \Pi x_1\colon \sigma_1.\ \cdots \Pi x_k\colon \sigma_k.\, \mathsf{Prop}$$

But this "higher type description" is artificial and requires auxiliary (and unnecessary) extensions.

Sums. Products Π in type theory are relatively easy, but sums Σ are much more complicated. They occur in various forms. As already mentioned, there are *weak*, *strong* and *very strong* sums. At this stage PTS's only deal with products, but a more complete theory of typed calculi will eventually have to deal with these sums as well. And it is then that it becomes important to have the dependencies that may occur, explicitly available in the formalism describing the type theory at hand. The differences between these three sums involve certain dependencies in the elimination rules. The formation and introduction rules are the same in all three cases. We describe them for $(\mathsf{Kind}, \mathsf{Type})$-sums.

$$\frac{\Gamma \vdash A\colon \mathsf{Kind} \qquad \Gamma, x\colon A \vdash B\colon \mathsf{Type}}{\Gamma \vdash \Sigma x\colon A.\, B\colon \mathsf{Type}} \qquad \frac{\Gamma \vdash A\colon \mathsf{Kind} \qquad \Gamma, x\colon A \vdash B\colon \mathsf{Type}}{\Gamma, x\colon A, y\colon B \vdash \langle x, y\rangle\colon \Sigma x\colon A.\, B\colon \mathsf{Type}}$$

Just like dependent $(\mathsf{Kind}, \mathsf{Type})$-products Π, these dependent $(\mathsf{Kind}, \mathsf{Type})$-sums Σ require a dependency $\mathsf{Type} \succ \mathsf{Kind}$. This is enough for the weak elimination rule:

$$\frac{\Gamma \vdash C\colon \mathsf{Type} \qquad \Gamma, x\colon A, y\colon B \vdash Q\colon C}{\Gamma, z\colon \Sigma x\colon A.\, B \vdash (Q \ \mathsf{where}\ \langle x, y\rangle := z)\colon C}\ (\text{weak})$$

But for *strong* sums one allows the above type C to contain a variable z of the sum type $\Sigma x\colon A.\,B$, as in

$$\frac{\Gamma, z\colon \Sigma x\colon A.\,B \vdash C(z)\colon \mathsf{Type} \qquad \Gamma, x\colon A, y\colon B \vdash Q\colon C[\langle x,y\rangle/z]}{\Gamma, z\colon \Sigma x\colon A.\,B \vdash (Q \text{ where } \langle x,y\rangle := z)\colon C}\ (\text{strong})$$

This occurrence of z in $C(z)$ shows that these strong sums only make sense in a setting with $\mathsf{Type} \succ \mathsf{Type}$, that is, with type dependency. In a next step, the *very strong* sums allow elimination as above, both with respect to types and with respect to kinds:

$$\frac{\Gamma, z\colon \Sigma x\colon A.\,B \vdash C(z)\colon \mathsf{Type}/\mathsf{Kind} \qquad \Gamma, x\colon A, y\colon B \vdash Q\colon C[\langle x,y\rangle/z]}{\Gamma, z\colon \Sigma x\colon A.\,B \vdash (Q \text{ where } \langle x,y\rangle := z)\colon C}\ (\text{very strong})$$

This very strong sum only makes sense in a setting where both Type and Kind may depend on Type, i.e. $\mathsf{Type} \succ \mathsf{Type}$ and $\mathsf{Kind} \succ \mathsf{Type}$. For example, we can now immediately tell that in polymorphic calculi with setting ③ only weak sums make sense (or are allowed, in the language of settings and features). And very strong sums may be added to the calculus of constructions (setting ⑦), but not to HML or the theory of constructions (with setting ⑥).

The point here is not that settings are there to forbid certain features, but to make clear right from the beginning (when one lays down the setting for a type system under construction) which features are allowed. The example of the various sums is meant to illustrate how much of an impact features may have on dependencies.

Notice that in case the sorts Type and Kind are the same, there is no difference between strong and very strong sums. There is a standard result which says that the very strong elimination rule can equivalently be replaced by a version with first and second projections. In presence of an axiom $\vdash \mathsf{Type}\colon \mathsf{Kind}$, the very strong sums lead (together with products) to a version of Girard's paradox, and thus to inconsistency.

One can similarly distinguish weak, strong and very strong equality in type theory. For equality the very strong version makes perfectly good sense in presence of $\vdash \mathsf{Type}\colon \mathsf{Kind}$.

Logic. Ordinary proposition logic may be seen as a degenerate form of predicate logic where there are no real predicates, but only closed ones (without free variables). Similarly higher order proposition logic—in which one can quantify over propositions as in $\forall \alpha\colon \mathsf{Prop}.\,\psi(\alpha)$—can be seen as a degenerate subsystem of higher order many-typed predicate logic, namely where one only has one single type Prop and no other (basic) types.

Under the propositions-as-types (and proofs-as-terms) point of view, a calculus of types and terms may be seen as a system of logic, in which types are viewed as propositions and a term inhabiting a type as a proof (or derivation) of the corresponding proposition. This raises the question: which calculi correspond to which logical systems? This has been a central topic of research in cubism (see for example [4]).

Since constants are usually ignored in cubism, some debatable correspondences are found. As an example we consider $\lambda\omega$. According to cubists, this calculus corresponds to higher order *proposition* logic. Formally this is entirely correct, but it only deals with a degenerate situation (as above). It is better and more natural to say that $\lambda\omega$-calculi correspond to higher order many-typed *predicate* logics. We write the plural form twice: there are various $\lambda\omega$-calculi and various higher order many-typed predicate logics and the variation is determined by the basic constants (that is, by the signatures $\mathcal{S}$) which are assumed to be given right in the beginning. (The setting here is already determined as $\mathsf{Prop} \succ \mathsf{Set}$, or, as $\mathsf{Type} \succ \mathsf{Kind}$.) Thus we have a λ-calculus $\lambda\omega(\mathcal{S})$ for every appropriate signature $\mathcal{S}$. And similarly in predicate logic: any standard text shows how one starts with certain basic function symbols and predicate symbols. Without the latter one does not get off the ground and one remains in proposition logic. The degenerate border cases on both sides (no constants) are related and only this point is mentioned by cubists. It's only the tip of the iceberg.

Explicitly, if a constant $\vdash \mathbb{N}\colon \mathsf{Kind}$ is added to $\lambda\omega$, then it suddenly becomes possible to form the kind $\mathbb{N} \to \mathsf{Type}$ of predicates on $\mathbb{N}$ and quantify over it as in

$$\Pi P\colon \mathbb{N} \to \mathsf{Type}.\, \Pi n\colon \mathbb{N}.\, \Pi m\colon \mathbb{N}.\, P(n+m) \to P(m+n)$$

which leads us into predicate logic. Thus, already one single constant brings down the cubists' correspondence between (pure) $\lambda\omega$ and higher order proposition logic. Restricting oneself to this $\lambda\omega$ without constants is a bit like limiting ones attention in the study of free groups to the singleton group (the free group on the empty set)[3].

One should see $\lambda\omega$-calculi (like other type systems or logics) as languages to reason about certain mathematical structures. It's like with groups: there one has three function symbols,

$$\vdash u\colon G, \qquad x\colon G, y\colon G \vdash m(x,y)\colon G, \qquad x\colon G \vdash i(x)\colon G$$

for unit, multiplication and inverse. When one reasons about a specific group G, then all the elements $a \in G$ are added as constants $\underline{a}$ to the language, and the resulting calculus is used.

There is the same picture for $\lambda\omega$. Constants of an appropriate mathematical structure may be organized in a signature $\mathcal{S}$. Then one can use the calculus $\lambda\omega(\mathcal{S})$ to reason about that structure. In $\lambda\omega(\mathcal{S})$ one reasons with explicit proof terms. One can also build a predicate logic on $\mathcal{S}$. Then one reasons without such proof-terms.

The purpose of this paper is to explain the ideas underlying an alternative classification of typed calculi in terms of settings and features. Nothing has been said about meta-mathematical or proof-theoretic aspects. But they can be formulated in the new language as well. For example,

(i) Let $\mathcal{L}$ be a type system such that for each dependency $s_2 \succ s_1$ in the setting one has (s_1, s_2)-products Π. Then Church-Rosser holds.

(ii) Let $\mathcal{L}^+$ be the following extension of $\mathcal{L}$ from (i): for each dependency $s_2 \succ s_1$ in the setting add weak (s_1, s_2)-sums. Then $\mathcal{L}^+$ is conservative over $\mathcal{L}$.

At this stage it doesn't matter whether these statements are true or not. They only serve to illustrate how one can do meta-mathematics. It is hoped that the settings and features described here enhance the understanding of type systems and clarify some of the issues involved.

In the end, what can be said about PTS's? I think the PTS-formalism works well for the *minimal* versions[4] of a *limited* number of typed calculi. But the formalism does

[3] This comparison ridicules matters in an unfair way: the free $\lambda\omega$-calculus on the empty set (that is pure $\lambda\omega = \lambda\omega(\emptyset)$ without constants) is not as degenerate as the free group on the empty set.

[4] And indeed, in the introduction to [1] it is clearly stated that only minimal versions are studied.

not scale up: not to extensions (like with sums or constants) and not to other calculi (like Martin-Löf's or HML and the theory of constructions). These are the technical defects of PTS's mentioned in the introduction. The conceptual defect is what we consider to be an inappropriate analysis of the structure of typed calculi: in PTS's the dependencies come out of the axioms and rules and are not taken as primitive.

Acknowledgement. Thanks to Herman Geuvers for helpful discussions.

References

[1] H.P. Barendregt, Lambda calculi with types. In: S. Abramski and Dov M. Gabbai and T.S.E. Maibaum (editors), *Handbook of Logic in Computer Science*, Vol. 2, Oxford Univ. Press, 1992, p. 117–309.

[2] A. Church, A formulation of the simple theory of types, *Journ. Symbol. Log.* **5**, 1940, p. 56–68.

[3] Th. Coquand and G. Huet, The Calculus of Constructions, *Inform. & Comp.* **76**(**2/3**), 1988, p. 95–120.

[4] J.H. Geuvers, *Logics and Type Systems*, PhD. Thesis, Univ. Nijmegen, 1993.

[5] B. Jacobs, *Categorical Type Theory*, PhD. Thesis, Univ. Nijmegen, 1991.

[6] B. Jacobs and T. Melham, Translating Dependent Type Theory into Higher Order Logic. In: M. Bezem and J.F. Groote (editors), *Typed Lambda Calculi and Applications*, Springer LNCS 664, 1993, p. 209–229.

[7] P. Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Napels, 1984.

[8] E. Moggi, A category-theoretic account of program modules, *Math. Struct. in Comp. Sci.* **1**(**1**), 1991, p. 103–139.

[9] D. Pavlović, *Predicates and Fibrations*, PhD. Thesis, Univ. Utrecht, 1990.

[10] D. Pavlović, Constructions and predicates. In: D.H. Pitt et al. (editors), *Category and Computer Science*, Springer LNCS 530, 1991, p. 173–196.

[11] W. Phoa, An introduction to fibrations, topos theory, the effective topos and modest sets, Techn. Rep. LFCS–92–208, Edinburgh Univ., 1992.